TU Clausthal

Clausthal University of Technology

An Approach and Design Pattern for Intra-Application Scheduling

René Fritzsche, Christian Ristig, Christian Siemers

IfI Technical Report Series IfI-10-11

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Federico Schlesinger
Contact: federico.schlesinger@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

An Approach and Design Pattern for Intra-Application Scheduling

René Fritzsche, Christian Ristig, Christian Siemers

Department of Computer Science, Clausthal University of Technology,
Julius-Albert-Straße 4, D-38678 Clausthal-Zellerfeld, Germany
rene.fritzsche@tu-clausthal.de
christian.ristig@tu-clausthal.de
christian.siemers@tu-clausthal.de

Abstract

This paper gives a comprehensive overview about a thread-based design pattern for embedded systems. This design pattern focuses on small-sized systems without operating system support, and a methodology for application-specific scheduling is given. This first step is enhanced by the approach to use time-related code constructs for automatic generation of this scheduling. This language enhancement is called time-enhanced C (TEC). To support the development as well as runtime, a monitoring concept providing a minimal-invasive approach is introduced as third part of this paper. This monitoring system, WatchCop, can provide the system as well as the developer with sufficient and precise information about the state of the program execution.

1 Introduction

The mainstream approach to design embedded systems is nearly almost based on von-Neuman-style microprocessor, independent of the size of the system. Additionally, it can be assumed that all systems – again independent of the size – integrate more than one task inside their application.

The first question is what size means or how size of an embedded system can be defined. The most common definition will use the intrinsic data size of the microprocessor, e.g. 8 or 32 bit. We think that this is an important parameter but by far not the only one, and therefore we chose an alternative definition based on the system's point-of-view.

For the purpose of this paper, we classify applications into two coarse-grain classes – called large-sized and small-sized applications. The classification is solely given by the presence or absence of an operating system. In practice, the use of an operating system enhances the resource requirements by a quantum leap. Of course there might be good reasons to use off-the-shelf operating systems like VxWorks, Windows CE, Linux and others, but we will show inside this paper that the simple requirement for scheduling capabilities does not force the use of it.

Therefore this paper concentrates on small-sized embedded systems – applications with multi-tasking capabilities but without use of an operating system. This approach is also useful for 32-bit-microcontroller-based systems, as computational power is often required – again with or without operating system.

We will introduce and discuss an approach for the design of such small-sized systems in connection with some new research approaches to support the design in embedded systems.

1.1 The Organisation of This Paper

The paper is organised as follows: To give a comprehensive overview about our approaches and the design pattern, we will discuss four major areas. First, a detailed definition of the term *function/unit thread* (f-thread) is given and discussed. As the most frequently used definitions of threads are fuzzy, even in systems supporting simultaneous multithreading (SMT) this is obviously necessary. The definition of f-thread and its implication to multithreading systems are discussed in section 2.

Succeeding then we will go into discussion how to design a multithreaded application for embedded applications and to prepare the required intra-application scheduling. This results in a design pattern which extensively uses the f-thread definition and simply provides the possibility of introducing a scheduling. The part can be found in section 3.

This design pattern is hand-made, and obviously the software designer has to handle it completely from scratch on. The question arises whether this could be performed automatically – at least partially. To address this requirement, we developed an approach called Time-Enhanced C (TEC). Programs written in TEC are standard C-code, therefore any compiler for C can translate them. So-called important comments are the major difference to C, and a pre-compiler capable of TEC will use them and transcode the complete source code into the required parts. The actual state of this research topic is discussed in section 4 of the paper.

Last but not least, we developed a monitoring system for development and runtime support. This system consists of a coprocessor directly coupled

to the main processor and uses a very small amount of additional instructions inside the program. Consequently the negative impact on runtime is not zero but remains very small and is in almost all cases negligible. This approach is called WatchCop and is described in section 5 of this paper.

Finally, we will give a summary about the developed parts and an outlook to future research in this area. Both can be found in section 6, and the paper concludes with some references.

1.2 Related work

As this paper combines several areas in research, related work can only be found according to the isolated topics but not for the complete paper. Here we divided the discussion of related work into several parts.

Design Pattern for Small-Sized Embedded System Applications

Specifically the support of small-sized embedded systems is still missing inside research literature as these systems seem to have no potential for new approaches. One approach can be found in [De04], where the design and the real-time behaviour of a small-sized microcontroller-based system are shown. The author uses an approach to classify the application parts into parts with (hard) real-time behaviour and without and mixes all parts during compile time.

The result is pre-defined runtime behaviour, and one of the most interesting parts is the approach to let a compiler do the task binding called Software Thread Integration. Any use of an operating system is avoided, but the approach lacks by the fact that the behaviour of the application must be precisely defined meaning that execution of program parts will use always the same number of clock cycles. This limits the use of this approach.

Time-Enhanced C (TEC)

The problem of scheduling tasks in an embedded system environment has been examined by many other papers before. There exist approaches of different task models like the multiframe (*MF*) model [Mo97] that extends the basic and well-known priority task model of Liu&Layland [Li73]. The MF-model describes tasks as a set of frames that are executed dynamically depending on their execution-times and given deadlines. For this approach the Worst-Case Execution Times (WCET) of all tasks need to be estimated first. If sufficient conditions are met, there exists a schedule for the given constrains. The drawback of this method is that tasks that are event-triggered will be regarded as periodic tasks with a certain, maximal

frequency of occurrence. The described approaches are of theoretical nature and are proven by using an operating system and adapted schedulers to the algorithms.

Approaches to enhance a programming language in order to implement constrain-statements have been discussed in [Le98] while other approaches defined completely new languages like *ESTEREL* or *Giotto* [He01].

WatchCop-Monitoring System

The monitoring of program execution is an issue since several decades. The authors in [Ma88] give a very good survey of approaches in the 1980s to take care of program execution. These early works are focussed on memory access errors – now a part of a memory protection system – and on illegal opcodes – inside modern architectures a task of the exception system. Most of the surveyed approaches in [Ma88] have found their way into the microprocessor and closely coupled units.

The approach in [Na05] on the other side addresses the actual requirements for reliability and real-time execution. The authors use a formal method to extract models from source code written in Ada for verification. The information obtained by verification is then used to instrument a specifically designed chip called SafetyChip. Real-time monitoring is performed by observing the communication between application and run-time kernel, and any deviance from the predefined behaviour is signalled to the system.

The authors in [Sa90] use an approach to implement monitoring functionality within a coprocessor for any program part, not only specific parts. The main constraint of this approach is to minimise runtime oberhead. To ensure this, the approach in [Sa90] as well as other approaches ([Ra05], [Fa08] and [Ir06]) use checksums and trace functionality to ensure or monitor the correct program execution in the sense of arithmetically correct program flows but not in timely manner.

In contrast to the briefly discussed approaches, WatchCop is used to monitor and control program execution concerning its timing behaviour first to ensure real-time functionality and secondly to check correct program execution at all. The main constraints are to minimise the impact on runtime – specifically by reducing the amount of additional instructions in the program flow – and the impact on processor microarchitecture – specifically by avoiding any impact on the architecture inside the execution pipeline.

Consequently, the WatchCop approach shows most similarity to the work published in [Na05]. In contrast to that, WatchCop may observe any program flow, even if a run-time kernel is not available, and a formal execution model is not required to enhance the design with monitoring capacity. Nevertheless such a model is very useful, and concerning

WatchCop, we are following the way to enhance a language model and a compiler to generate the monitoring information semi-automatically.

Compared to all previously discussed approaches, WatchCop shows the following differences:

1. WatchCop monitors and partially controls the timing behaviour, not just the control flow behaviour of the program and couples runtime and real-time.
2. With just a minimum of negative impact on runtime behaviour and maintaining the original processor architecture, WatchCop may monitor all kinds of program flows with no restriction.

2 The Definition of an F-thread

Since we discuss a multitasking approach inside one application without operating system support, this will be necessarily a multi*threading* approach. The running application will be the only process inside the system, and this process will contain several threads.

The common definition of a thread is often something like a *light-weight process*, or a *self-containing program part with a minimal private context owned by the thread and with defined communication interfaces*. Based on these fuzzy definitions we define the *f-thread* as follows:

An *F-thread* (Function/Unit Thread) is a process part performing a well-defined task with a minimal private context owned by the thread. This process part communicates with other parts of the process by well-defined interfaces, and if the communication partners do not belong to the same f-thread, communication is only performed indirectly without synchronous coupling between these parts.

Apparently, coupling between f-threads has no direct timing constraint. Inside one application using imperative languages like C, an intra-application f-thread is implemented as a high-level function, and of course this high-level function can call other (lower-level) functions directly.

Naturally this is allowed, even if the low-level functions are shared by more than one f-thread, but it is strictly prohibited to call another f-thread inside the same application by such a direct function call – this would be certainly synchronous coupling. The communication must be message-based, meaning that the message-sending f-thread puts its message into a communication queue, where the message queue controller will check it later and will call the corresponding receiving f-thread with this message.

The advantage of this – in comparison to commonly used thread definition more limiting - f-thread definition is that the message queue controller is automatically invoked between two f-threads, and this is a very simple but effective scheduler. Additionally the system might be designed on system level when mapping parts of it on hardware or software is still open. If the system is structured by f-threads during design time, and the f-threads will follow the above given definition, it is very simple to map any f-thread into hardware, if timing or other requirements will force the designer to do so.

In this case, communication is not performed by the message queue and the scheduler directly, but hardware buffer and an interrupt-service-routine based communication detection will perform the coupling as shown in section 3. In this sense, the multithreading approach with indirect communication via message queue appears to be an ideal base for hardware/software co-design approaches.

The implementation of f-threads as (high-level) C-functions or as hardware units was responsible for the chosen name function/unit thread.

3 The Design Pattern for Application including Multithreading

This section gives an introduction to the design pattern which uses the aforementioned f-thread definition and integrates events from outside the system with intra-application events and messages. For the rest of this section it is assumed that the application is well-defined concerning its algorithmic behaviour and its timing requirements.

3.1 The Steps of the Design Process

Step 1 consists of structuring the application into f-threads. This is a very important part of the design process, and for complex applications it may be necessary to repeat this step several times to meet all requirements of the system.

The boundary of an intra-application f-thread is in first order defined by the algorithmic content. The design will always tend to integrate one task in its complete semantic into one f-thread with the goal to minimise communication with other f-threads. As an example, one f-thread could perform the complete decoding of incoming messages from a network.

Minimising communication corresponds to maximising the content inside one f-thread, but there will be a second boundary concerning the timing behaviour of the f-thread. Inside our model, a *cooperative multithreading* is used, and this means that any f-thread must strictly follow timing constraints.

This design part is much harder to perform, because timing behaviour is hard to predict and even harder to proof. Nevertheless we assume that runtime estimations with sufficient precision exist and that an allowed maximum runtime per f-thread without interruption is also known. This time is something like the maximum possible time for an f-thread to run continuously without returning to the scheduler, and often this time is easy to define but hard to estimate. We will address this issue again later.

If the estimations show that some f-threads will use more time than possible or allowed, these f-threads must be partitioned enhancing the communication overhead. Normally this communication overhead is negligible, but in specific cases this must be balanced. Partitioning means that the original f-thread is partitioned into two or more parts. Each new part forms a new f-thread and must observe the indirect communication rule.

If, for example, the decoding of a received message is partitioned, each partition could now correspond to one communication layer. Intra-application communication could use a shared memory for the received packet and messages from prior f-threads to the succeeding.

3.2 The Design Pattern for the Embedded System

Fig. 1 shows the resulting software structure. The complete application is partitioned into f-threads, and a scheduler exists which manages the communication between all parts of the system.

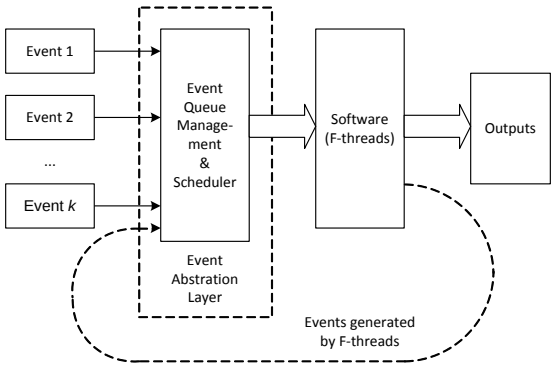

Figure 1. Event-based scheduling and communication flow

The communication flow is explained now in detail. Incoming events from hardware, where the first reaction might be implemented e.g. as interrupt service routine, form the first class of inputs into the communication scheduling. The second class are messages generated by the f-threads, which are also put into the message queue. Listing 1 shows the management of the message queues.

This part of the application performs at least two important tasks:

- The abstraction of the (more hardware-related) events and the (software-related) message to a unified message queue offers the advantage that all events/messages can be scheduled uniformly.
- This part is capable of performing an application-specific scheduling

```
int iStoreNewEvent( int iEventType, int iEventData )
{
    if( stcEventList[iEventWrite].iEventType != EVENT_NO_TYPE )
    {
        return( 0 );
    }  /* The return value 0 means that the list is filled  */

    else
    {
        stcEventList[iEventWrite].iEventType = iEventType;
        stcEventList[iEventWrite].iEventData = iEventData;

        iEventWrite++;
        if( iEventWrite >= NUM_OF_EVENTS )
            iEventWrite = 0;

        return( 1 );
    }  /* 1 means the successful storage of the event  */
}


/*****************************************************************/
void vGetNextEvent( struct stcEvent *stclTemp )
{
    if( stcEventList[iEventRead].ui8EventType != EVENT_NO_TYPE )
    {
        *stclTemp = stcEventList[iEventRead];
        stcEventList[iEventRead].iEventType = EVENT_NO_TYPE;
            /* Memory is freed  */

        iEventRead++;
        if( iEventRead >= NUM_OF_EVENTS )
            iEventRead = 0;
    }

    else
    {
        stclTemp->iEventType = EVENT_NO_TYPE;
    }  /* No event available  */
}
```

Listing 1. Message/Event Queue Management

To unify events and messages, it is necessary to define a frame for the interrupt service routines (listing 2). The implementation of this routine contains only a minimum of algorithm, the rest is mapped to the f-threads and the event scheduler. This rule has to be carefully observed, otherwise the abstraction layer given by the message/event queue management is

incomplete. For the rest of this paper, events and messages are used synonymously.

```
void interrupt vISR()
{
    int iISRRegister;
    iISRRegister = iGetISRRegister();
    if( ISR_FLAG1 == (iISRRegister & ISR_FLAG1) )
    {
         iStoreNewEvent( EVENT_1, 0 );
    }    /* Pseudodate 0  */
    else if (…)
         …
}
```

Listing 2. Frame for interrupt service routine to map hardware events into software

Last not least, the event/message queue management is described. Listing 3 shows some rudimentary code for describing the principle. The main loop consists of an infinite loop, where the next event is received from event queue and an according f-thread is invoked. If no event or message is available, nothing happens.

This part is something like a very basic scheduler serving the First-Come-First-Serve-strategy. At this point other strategies can be integrated, and because this is application-specific, the best strategy can be developed.

```
main()
{
...
    while( 1 )
    {
        vExecuteEventList();
    }
}


void vExecuteEventList( void )
{
    struct stcEvent stclTemp;

    vGetNextEvent( &stclTemp );

    switch( stclTemp.iEventType )
    {
        case 0:
            Compute_for_Case0();
            break;

        case 1:
            Compute_for_Case1();
            break;

        default:
            break;
    }
}
```

Listing 3. Event management (scheduler)

3.3 Other Design Aspects

As a preliminary resume, this design pattern consists of the identification of f-threads and their timing requirements, eventually a refinement of the f-threads, the implementation of these f-threads as independent (high-level) functions and the construction of a scheduler according to the requirements of the application. But there is another aspect of this design pattern to be remarked here.

The mapping of a f-thread on a high-level function and therefore in software will be the mostly used way, but when the timing constraints will be too hard, the mapping on (e.g. programmable) hardware is also possible. The definition of an f-thread in section 2 includes also this case illustrated in figure 2.

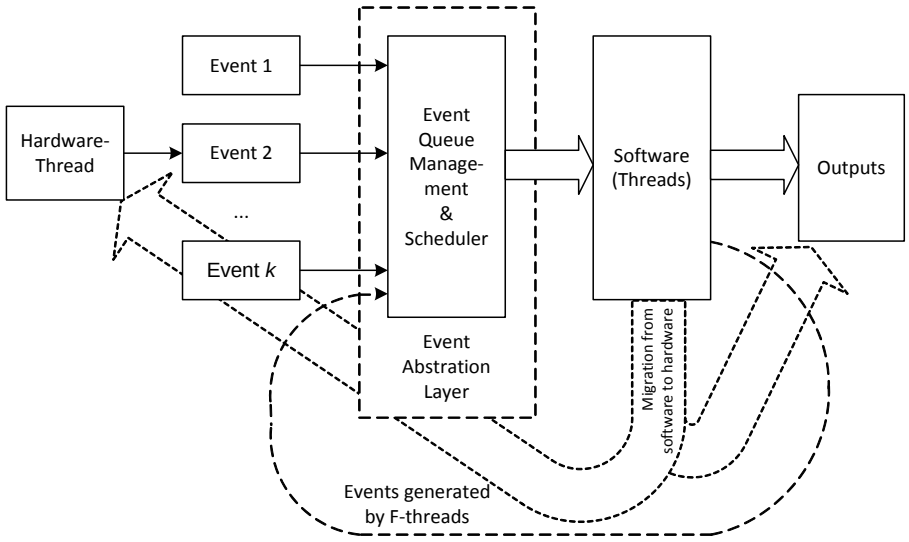

Figure 2. Migration of an f-thread from software to hardware

In practice, the newly in hardware implemented f-thread will be located at the event input and at the output area as shown in figure 2. This means that the basic architecture is still microcontroller-dominated, while parts of the application with very short timing constraints are mapped on hardware. This is the most satisfying model for embedded systems.

4 An Approach for Automatic Generation of the Scheduling

Section 3 has introduced the design pattern for small-sized, microcontroller-based embedded systems capable of running several f-threads. Any concrete design is up-to-now hand-made, therefore the question arises to automatically generate at least parts of it. This approach stays and falls with the availability of timing constructs inside the program.

The here presented approach to enhance the program code with timing constructs and constraints is a simple extension of "C" called *Time-Enhanced C (TEC)* and was first introduced in [Fr08]. The approach can be assigned to any other imperative programming language that supports comments or injection of additional timing constraints.

In the following subsections, the workflow using TEC are discussed in section 4.1 followed by the syntax elements in 4.2, as they are defined up to now. Section 4.3 contains a brief description of transcoding approaches actually under research.

4.1 The Workflow using TEC

The information provided by TEC-statements is extracted by a precompiler in order to set up a scheduling-template that uses parts of the original code. *Coroutining* is applied to the functions so that even micro-f-thread-scheduling can be performed. After this, a new main-function containing the initialisation, basic start-up code and a way to call each of these functions/f-threads must be generated using the given timing information and dependencies. This describes a more or less static scheduling algorithm, but we will show that there are still ways to react on events and non-predictable interrupts from I/O-devices.

4.2 The Syntax of Time-enhanced C

The extension of C allows specifying which function or task has to be executed at what time. Time may be given as periodic statements declaring when to initiate a task and also implicitly declaring a deadline for this. But also simple information about the explicit deadline may be specified. Another way of modelling the function flow is by defining dependencies of task ending and task starts.

To support any further scheduling scheme with additional timing information or program flow constraints TEC offers the possibility of using

special comment-lines directly in the C-Code. These fit into the following structure and therefore are easily extracted and processed by the according precompiler to generate the new main-structure of the program:

```
/* @_TEC_@ statement, statement, ... */
```

Where multiple “statements” represent the various information types TEC provides. The preliminary defined statements are:

-periodic <time>
: This specifies a repeatedly executed function with given period. This is the common and usual type of function to be scheduled. If a *duration*-statement is not included, the deadline of this function is set to the length of the given period.

-handler <event>
: marks this function as an event-handler. This function will be never switched off is executed every time the given event occurs. No other additional statements can be specified.

-after <event>
: This is the trigger of a function with respect to a given event/flag. It works similar same as the handler-statement but the task may be switched off if necessary due to shortage of time.

-once
: This defines a one-shot execution - used for initialisation- and setup-threads. Functions will not be part of the scheduling scheme but called directly before the main-loop.

-start_time <time> & *end_time <time>*
: This is the definition of the invocation time and maximal execution time of a function. A given *end_time* declares a deadline for the task.

-duration <time>
: This statement specifies the information about the WCET of this function. The information will be used inside the schedule.

These statements will be used to schedule the f-threads. The information is not mandatory – any function specified will be treated as be executed only once and the deadlines will not be set up. Missing *start_time* or *end_time*-statement will be substituted by the *periodic* times and as described the deadline will be set to the length of a period if no *duration* is given.

4.3 Transcoding C-Code driven by TEC-Statements

The next step inside an automated design flow is to build the transcoded source and to integrate scheduling inside. The necessary condition to perform this for a given application is that f-threads included in scheduling must be either short enough to be executed within one (virtual) time-slice of the schedule or have to be split into subtasks using the coroutine-approach.

Please note that the time-slice is not real, because scheduling is cooperative and not time-slice oriented. In fact it is necessary to assume well-behaved f-threads in the sense that they return to scheduling within a hard deadline to obtain real-time behaviour. Our approach now is to automatically partition an f-thread if WCET estimation shows values above that deadline.

4.3.1 Generation of coroutines

Routines that allow cooperation with other f-threads by providing special exit- and re-entry-points are called coroutines as described e.g. in [Si07]. For scheduling purposes, functions containing long-running loops need to be split into shorter subtasks or need to be able to preliminarily exit these loops and continue at the same place without losing intermediate results. The following method is able to transform exemplarily any *for*-loop into a coroutine-valid structure (listing 4).

- Make all used variables static (which in fact means they will behave like global-variables with visibility limited to the function).
- Transfer the initialization-statement of the loop to a common variable definition section.
- Insert an exit-point to the loop, which terminates the loop after a certain amount of repeats (e.g. exit after every loop).
- Return values are stored in global, volatile variables – thus return-statements are obsolete.

With this approach functions are forced to fit into cooperative scheduling, as long as the original function consists only of this for-loop. In some other cases, when the function to be transcoded holds more content, this can be also transcoded automatically but with more effort. This is currently under research.

Extending the approach (listing 4, step C) by extracting the loop-statement and moving to the calling function allows to estimate execution times at runtime as now all used variables and dependencies are available in the main-function. This method will be used in following procedure.

```
int func()                    volatile int x=0;              volatile int x=0,i=0;
{                             void func()                    void func()
   int i,x=0;                 {                              {
   for (i=0;i<10;i++)            static int i=0;                x=x+i;
   {                             if (i<10)                      i++;
      x=x+i;                     {                           }
   }                                x=x+i;                    // -- main function--
   return x;                        i++;                      while (true)
}                                }                             {
                              }                                  if (i<10)
                                                                    func();
                                                                 …
                                                              }
      (A)                           (B)                            (C)
```

Listing 4. Transformation of a simple function into a coroutine (steps A to B), moving the loop-statement to the calling function (C)

4.4 Scheduling

Implementing a scheduling directly into the application code means setting up a structure for calling function parts (frames, subtasks) including switching through all active tasks that have to be processed. Parts of functions can be obtained by using the coroutine approach and by modelling the code as state machines and transforming them state-by-state back to single functions that contain only sequential code. The complete flow-control will then be done by the scheduling template.

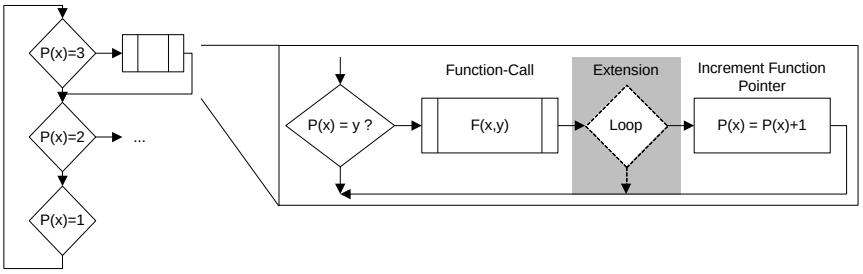

Figure 3. Template of sequential execution of frames F(x, y) of f-thread x

Figure 3 shows the usage of a template capable of calling frame by frame of one f-thread and additionally providing f-thread-switching. This is achieved by executing one frame of a task followed by checking other subtasks of different f-threads to be executed before calling the next frame. That is why the order of function calls is reversed – thus the program-flow is lead through the complete outer loop-structure of the calling routine before re-entering the task.

Frames of task *x* are selected through a function-pointer *P(x)* which will be incremented each time a function part *F(x,y)* was successfully executed. To initiate a task the corresponding function-pointer just needs to be set to 1. An advantage of this procedure is that interrupts that usually trigger event-flags only need to set a variable to 1 in order to start the corresponding handling tasks. This structure forces tasks to cooperate with others as the executing unit has to check other tasks for ready frames before focusing execution back on the first one. This approach is called *forced cooperative design*.

The extension of this structure will be used if loops inside the given function were abstracted to the top-level scheduler as described in section 4.2.

4.4.1 Main-Template

At this point a way for assembling a main routine containing all tasks being transformed to a multi-frame-set as shown is presented.

Tasks in this template require a function pointer *P(x)* and a flag to define whether it is an active task or not. Lining up all subtask calls of all tasks without this flag would lead to a design where all tasks are executed repeatedly and shorter tasks would emerge more often than longer ones.

The presented schedule ensures that tasks are forced to process from time to time but deadlines are not included to this structure (figure 4). The priorities of tasks are mapped to the position of the task-block within the main-loop – subtasks of tasks at the top are executed earlier than others.

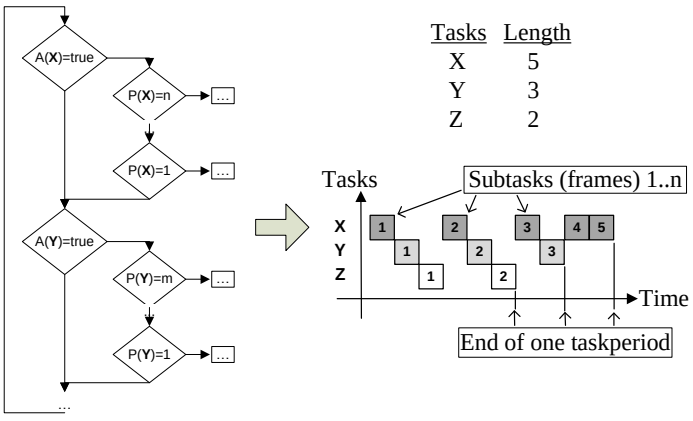

Figure 4. Main-loop and example of scheduling resulting

4.4.2 Selection of tasks to be processed

An important improvement is reached by implementing a priority algorithm that selects the next tasks to be processed each time a subtask was called and has returned. These points within the schedule will be called *super-synchronization-points* (SSP).

The selection algorithm needs to figure out which task to be executed next in order to hold its deadline. This information has to be obtained at runtime as event-driven tasks can use more processor-time than expected at compile-time. By this reason a semi-dynamic algorithm has to be implemented that supervises all deadlines and active tasks.

Earliest deadlines first (EDF) scheduling is capable of fulfilling given deadlines but fails if there are too many non-predictable events to handle – rate monotonic scheduling (RMS) reduces the impact on the most-important and therefore high-prior task. A mixture of these two approaches takes the upcoming deadlines and the non-detachable tasks into account:

Let x be a task, $L(x)$ the number of frames of x, $F(x,y)$ a frame y of task x, $E(x,y)$ the execution-time of frame y, $D(x)$ the given deadline and $P(x)$ the function-pointer that points to the frame to be executed, $A(x)$ the active-flag and T the set of schedulable tasks and N all non-maskable tasks.

The time required by all remaining frames of task x is given by:

$$R(x) = \sum_{i=P(x)}^{L(x)} E(x,i) \tag{1}$$

Remaining time until deadline of task x from now (time t):

$$r(x) = D(x) - t \tag{2}$$

$$d = r(x) - R(x) - \sum_{\substack{j \in T-\{x\} \\ A(j)=true}} E(j, P(j)) - \sum_{j \in N-\{x\}} R(j) \tag{3}$$

d represents the time required by the actual task x to finish using the basic approach of running all tasks. If this value is below 0, a single switching through all tasks would waste more time than available till the deadline of x. At this point all other maskable tasks should be deactivated and only the current tasks should be processed in order to not miss the deadline. This enabling and disabling of tasks can be done by using their corresponding active-flags. Permanent tasks, always enabled (=non-maskable, without active-flags), will still be processed and therefore input handlers or tasks needed for safety reasons won't be harmed with this approach. Equation (3) gives an indicator, when to switch from walking through all tasks and single-task-mode and thus providing a high priority.

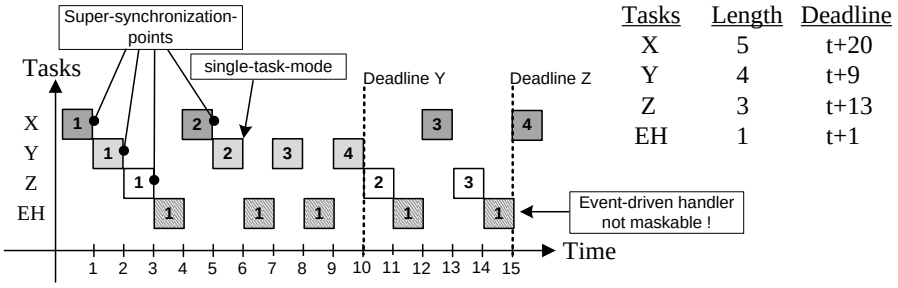

Figure 5. Improved interleaving scheduling structure with selecting single-task-processing at super-synchronisation-points

An application of this approach is shown in figure 5. At each Super-synchronization-point the sum for R(x) and d are calculated and if necessary as shown at the 6.timestep tasks X and Z have been disabled leaving Y active in order to fulfil the given deadline. The Event-handler is of course still processed and thus interrupts will be handled and not ignored during this single-task-mode.

The calculation of the given equations takes more and more time the more tasks have to be observed and given deadlines have to be taken into account. With the approach of using an additional hardware build next to the main CPU that is able to surveil the timing online and may also sum up task completion ratios independently and with less impact to the runtime. This attempt to introduce such an unit will be described next.

5 Monitoring the Design with WatchCop

The use of the introduced design pattern and the TEC-enriched code with automatic (or manual) transcoding results in software showing complex runtime behaviour. This demands for development and/or runtime support to ensure correct behaviour. It may be performed by monitoring support, which is required during development, specifically during test phases, and which may be additionally required for scheduling support. The last case is the most interesting one: The availability of scheduling support by a coprocessor in combination with no negative runtime impact would result in a very precise, inexpensive scheduling.

The here discussed approach is called *Watchdog Coprocessor* or shortly *WatchCop*. It consists of a hardware/software interface using a small set of additional instructions, a set of timer and a set of internal registers or internal memory for monitoring timing behaviour. This architecture is discussed in detail now.

5.1 Basic Architecture

Figure 6 shows the basic architecture of WatchCop. A similar approach to [Sa90] is used in the sense that all functionality is implemented within a coprocessor with a minimised impact on program execution time.

The architecture shows two major parts, containing the basic part and the extended part. The basic part carries all necessary parts for monitoring applications including an interface to obtain the monitored values. This addresses operating system issues in the sense that actual runtime data are available for use in scheduling decisions.

The extended part on the other side contains direct control mechanisms and all necessary interfaces. This was designed to directly control runtime behaviour and to recover from exceptional operating conditions like deadlocks. This may be used to control program flow as well as to support operating system capabilities.

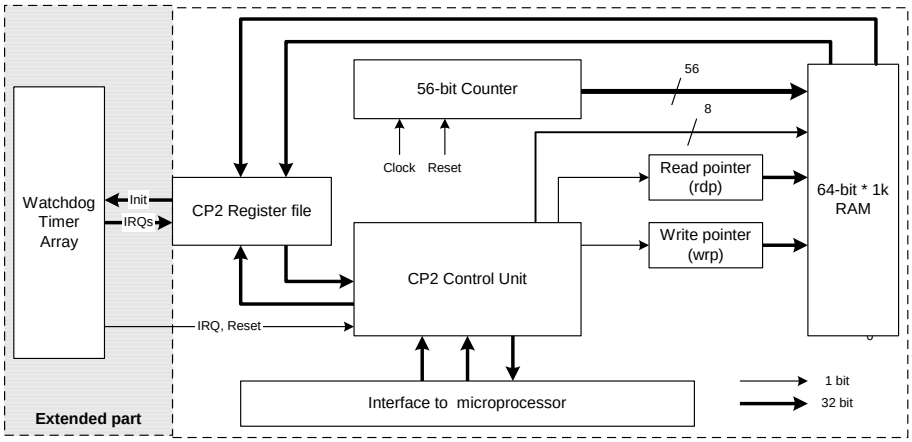

Figure 6. Basic/extended (left) microarchitecture of WatchCop

The shown implementation in figure 6 (without the shaded extended part) contains a free running counter of arbitrary width (here 56 bit), a set of 4 coprocessor registers, a read and a write pointer to manage the ring buffer for storage of monitoring events, the monitoring memory of arbitrary size (here 1K depth with 64 bit width) and the control unit. While the arbitrary values were chosen for long-running counter implementation without overflow, maintaining the other 8 bit for label identification and for easy interfacing the internal 64 bit with a 32-bit microprocessor, these values may change frequently.

The basic approach works as a monitoring system. If the processor fetches a cop2-instruction with according arguments, this instruction is directed to the coprocessor and executed there. In detail, 8 bit of the

argument are decoded as label number and stored together with the actual 56-bit counter value inside the RAM. This implements the monitoring functionality, while the read and write pointers manage the access using specific cop2 instructions.

5.2 Extended Architecture to implement Watchdog Functionality

The monitoring capacities of the basic architecture are extended by some watchdog functionality (see fig. 6 including the extended part). For this purpose, a set of configurable counters is integrated in the architecture. Each counter may be configured independently to a start value and to a label number. During program execution, each cop2-instruction containing this label (ref. section 5.3) as argument resets the corresponding counter to the start value.

On the other side, the counter is continuously decremented each clock cycle, and if an underflow occurs, a signal to the processor is activated, because a defined runtime condition was not met. This is well-known watchdog functionality with the extension that several points in program flow might be used to monitor the runtime behaviour.

Different to known watchdog implementations, the severity level of this watchdog alert is configurable and may change between two successive events. WatchCop uses at least two signalling lines to the processor, one for interrupt request, the other for reset. Therefore, a watchdog timer underflow can initialize an interrupt, e.g. if this is the first time, and may reset the processor, if this happens again. Applications of this feature are discussed in section 5.4.

5.3 Hardware/Software Interface

The interface between WatchCop and the application software contains three parts (see also figure 7): The coprocessor instructions for configuration during initialisation phase and for monitoring, the signalling part addressing interrupt service resources of the processor, and the data exchange for initialization and monitoring evaluation. This reflects to the four main parts of software interface between main program and monitoring: Initialization, monitoring during program execution, reaction on event signalling and monitoring evaluation.

During initialisation, the main processor may set several register to arbitrary values for configuring the coprocessor. After this period, the coprocessor normally works autonomously without executing an

instruction flow provided by the main processor. Only few instructions are inserted into normal program flow to synchronise the coprocessor with instruction flow of the main processor. This interface was designed to reduce the impact on program execution as much as possible while maintaining the coprocessor approach to preserve the main processor from redesign, as mentioned before.

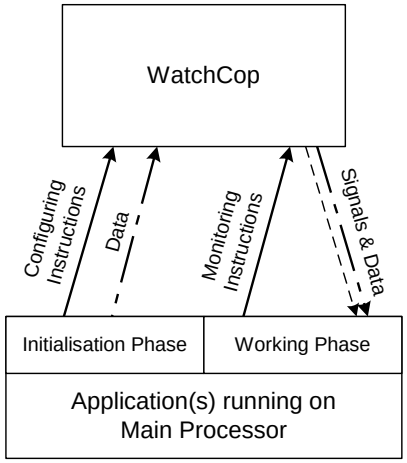

Figure 7. Interfaces between WatchCop and main processor

5.3.1 Hardware Interface between WatchCop and Main Processor

The hardware interface to/from coprocessor and main processor consists of a bus system for instructions of arbitrary size, e.g. 32 bit width, optionally of a data bus system of arbitrary size, and of few signalling lines. While the instruction bus system is mandatory, the same lines could be used for transmitting additional data, if they are capable of bidirectional data transfer, when data transfer from WatchCop to main processor is desired. The latter could be required for transmitting monitored values from WatchCop to main processor for further evaluation.

The signalling part was designed to immediately inform the main processor about events like watchdog timer underflow. During first project evaluation it appeared to be desirable to implement a signalling system with more than one level, and a two-level system was chosen. The 1st and 2nd level signalling is mapped on interrupt requests of arbitrary priority. In most cases, the 2nd level underflow of one watchdog timer will generate a highest priority interrupt request, in many cases of non-maskable type.

5.3.2 Instruction Set

The chosen instructions were picked from the standard instruction-set of MIPS-based microprocessors in order to be able to use unmodified development tools and compilers. Due to this fact language-checkers and optimizing strategies may still be applicable after inserting our monitor-instructions to the code. Three commands are necessary for using all WatchCop functionality:

- The mtc2-instruction (move to coprocessor 2 register) copies data from processor register to a specified coprocessor register. This may be used for initialization of some functions, e.g. timeout values for the watchdog timer or all kinds of configuration setup.

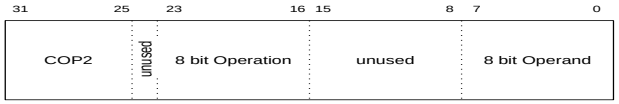

Figure 8. Binary format of COP2-instructions in WatchCop

- The mfc2-instruction (move from coprocessor 2 register) copies data from coprocessor register to processor register. This will be frequently used, e.g. for reading status register or accessing monitored values in the event list.
- The cop2-instruction (coprocessor 2) is used as general purpose format for all other instructions. This format normally holds some bits for free use, in our implementation 25. Figure 3 shows the usage of the bits: Bits 23 through 16 contain a function number, bits 7 through 0 a label number.

The list of actually implemented functions for cop2-instruction is shown in table 1. Actually this may be extended for additional functions if required, because the instruction set space is not densely used in the current implementation.

Table 1. Used subformats for some cop2-instruction

Operation	Operand	Descrpition
No_Operation	None	Do nothing
Reset_Monitoring	None	Monitoring is stopped, all pointer are reset
Start_Monitoring	None	Monitoring is (re-)started, from this point on every Store_Event is recorded
Stop_Monitoring	None	Monitoring is stopped, but data are maintained
Set_Read_Pointer	None	Sets the read pointer to first/last value
Get_value_from_list	None	Gets the value from the list in RAM pointed to by the read pointer.
Increment/decrement read_pointer	None	The read pointer is incremented/decremented (in circular way)
Store_Event	Label (8 bit)	The actual clock counter is stored in RAM including the label
Set_Timeout_Value	Watchdog Timer (8 bit)	The 56-bit value in CP2 R2/R3 is copied into the according watchdog timer

5.4 Applications with Use of WatchCop

5.4.1 Applications Using the Basic Architecture

The first major application that is supported by WatchCop is the development, specifically the test of real-time applications. For this purpose, the critical paths or parts to be tested will be instrumented by cop2-instructions with monitoring. Each time the program runs through this point a label containing the 56-bit counter value is written into the private RAM of the WatchCop, and may afterwards be read and analyzed for real-time behaviour.

As 256 different labels may be inserted into assembler code, some interference like “if the program runs through label xx, timing constraint on label yy is not met” between paths through the program may also be observed. This leads to a post-run analysis with exact timing labels with very low impact on the runtime (by the additional instructions). Using this basic feature during runtime is also possible and a good monitor to prove real-time behaviour, even after problems have occurred, or as information base for operating system decisions.

This approach was successfully tested during testing real-time applications using several f-threads to monitor specifically inter-thread communications. These communications, either implemented in blocking (waiting) or non-blocking fashion, show runtime-specific and even data-specific behaviour. Therefore the observation of realistic behaviour might be essential to consider modifications to the design, and it is hard work to realise these realistic constraints only by simulation.

5.4.2 Applications Using the Extended Architecture

The full power of WatchCop is provided by the extended features. This is due to the fact that WatchCop has now a direct feedback channel by using signalling lines for interrupt request or reset, and therefore watchdog, challenge/response-functionality and extended support for scheduling may be included into the application.

5.4.2.1 n-Level-Reaction Scheme for Watchdog Events

First, the watchdog part inside can be used for a classical watchdog functionality. Up to 8 timer in the actual implementation can be configured as watchdog timer and may be started. Specific cop2-instructions will reset the timer to its initial value, if the program executes this instruction. Therefore, WatchCop supports 8 different watchdog activities during runtime.

We implemented a configurable 3-level reaction scheme. The first underflow of any watchdog timer results in an interrupt request, a highly prioritized interrupt/trap or in a reset. The level of reaction is configurable, and therefore the 'classical' functionality of a watchdog might be configured. After requesting the interrupt, an additional time is loaded into the watchdog timer register. This time must be explicitly configured during initialization, otherwise it is set to '0', and the next level of reaction is instantaneously invoked.

If the processor reacts during that additional reaction time, the system appears to work properly, and the watchdog is reloaded with the original value. If not, the second reaction level is started and results in a trap (non-maskable interrupt) or a reset (if additional time is configured to 0). The third level is always a reset, but in other implementations even more level might be integrated. In this case the processor has run out of control, and the application must be restarted completely.

This may be used for additional functionality in the following way. If the processor receives the 1st-level interrupt request, the reaction inside the interrupt service routine (ISR) might be the reset of the watchdog timer. Nevertheless it is still not sure that the processor does not stay in a deadlock inside the main-program, because the reaction is performed inside the ISR.

To avoid this misinterpretation and to introduce a challenge/response-function, it is proposed to include the reaction not into the ISR but into normal program operation. The interrupt initialised by the coprocessor results into a software event which in turn must be handled inside main program (see listing 5), and algorithmic capacities may also be included. In this case, the interrupt is interpreted as challenge, and the software of the processor responses.

```
void interrupt vISR()
{
   ...
   switch( cp2IRQStatus )
   {
      case WATCHDOG_LEVEL_1:
         setEvent( EVENT_WATCHDOG_L1 );
         break;
   ...
   }
}
```

```
void main()
{
   int tempEvent;
   ...
   while( 1 )
   {
      getEvent( &tempEvent )
      switch( tempEvent )
      {
         case EVENT_WATCHDOG_L1:
            resetWatchdogTimer();
            break;
   ...
   }
}
```

Listing 5. Code fragment to integrate challenge/response-functionality

5.4.2.2 Scheduling Support by WatchCop

Scheduling is basically supported by measuring time differences inside WatchCop. If the microprocessor system does not use an operating system, the possibility of using a cooperative approach for f-threads with an application-own scheduling as described in section 3 might be used. If all cooperative f-threads work perfectly well and do not block, everything is okay, and the system works with high efficiency. All overhead from an operating system is omitted, but multithreading is still supported.

If the application is not as perfect as described, WatchCop can be used to obtain a good compromise between cooperative and preemptive scheduling. The basic principle can be still cooperative, but all f-threads are individually controlled by the n-level watchdog mechanism. In this case the planned reaction upon a watchdog timer underflow could be the cancelling of the blocking f-thread (which is definitely a serious issue) and continuing work with next event or next f-thread.

This scheduling was successfully implemented, and we call it forced-cooperative due to the fact that preemptive scheduling only occurs when exceptional operating conditions are observed and f-threads block. This is comfortably supported by WatchCop, even if the forced scheduling times vary from f-thread to f-thread.

Furthermore, the TaskCombining approach as described in [Si05] or the TaskPair approach in [Ge01], both are supported. In this case a twofold reaction scheme for real-time applications is proposed. If reaction time is tide, at least one of a set of tasks is not executed but reacts with an emergency value. This reaction system is known as precise time, imprecise logic.

As shown in [Si05], the presence for timer support is essential for efficient implementation. In this case, the WatchCop may support with the built-in timer, because they can support the processor with according interrupts. One watchdog timer is used for each task inside the task-combining-system and is set to a value close to the deadline of this task with a reaction value of “trap”. If this value is reached without reset before, then the task is not able to react precisely, and the imprecise value (which must be computed earlier as discussed in [Si05]) is used. As long as this task has not started to perform computation and all others are ready or close to readiness, no time is wasted.

6 Summary and Outlook

The purpose of this paper was to give an overview about our approaches to embedded system design. These approaches were the f-thread-based design pattern (including the f-thread definition and the built-in co-design), the language extension time-enhanced C for automatic generation of multithreading-capable systems and the monitoring system with scheduling support.

The first part, the f-thread-based design pattern, uses an asynchronous communication model between two different f-threads enabling a scheduling of the f-threads. In combination with the event abstraction layer translating all interrupting events into software events or messages, this model unifies the performing of all events and appears to be universal for small-sized microcontroller-based systems.

These advantages are very useful but come at the price of a more complicated hand-made design. This is the initial point for the second part, time-enhanced C, which is a language extension for C (or other imperative languages). The idea is that additional time-related constructs inside the code will enable a specialised pre-compiler to automatically generate the f-thread/scheduling system from the source code.

This system for automatic code generation is currently under construction and research.

For the monitoring approach we presented a system consisting of a coprocessor specialized to monitor execution time as well as a rudimentary software interface to use this system. The purpose is to implement a reaction system for deadlocks and other software errors to keep the software-based system operating.

This WatchCop is implemented as softcore using a MIPS32-compatible processor with coprocessor interface. The implementation of the coprocessor uses roughly 20% of the processor, with most of the silicon area is used for storing time/label values in the RAM. These values are the base for detecting time failures or weaknesses and for correction. As simple example, an operating system providing preemptive scheduling using the WatchDog capabilities could be implemented very easy and fast.

The next step will be to integrate the coprocessor into high level languages and to establish high-level language support. The roadmap to do this is to use normal C and to enhance this with special comments that are interpreted by a pre-compiler. The pre-compiler extracts the timing constraint from source code and generates a user constraint file, which is then interpreted for code generation.

This approach is first published in [Fr08] and [Fr10], and WatchCop is the ideal hardware architecture to execute the necessary monitoring and

measurement for inter-thread scheduling. The formal models in [Na05], which are essential for defining the correct functionality of the proposed Safety Chip, are here replaced by language constructs.

While the integration of high-level language support and WatchCop into a development tool and framework is the actual next step, the plan is to automatically generate a multithreading system from such enhanced C-code including WatchDog capabilities to support multithreading and monitor the complete system.

References

[De04] *Alexander G. Dean*, "Efficient Real-Time Fine-Grained Concurrency on Low-Cost Microcontrollers". IEEE Micro 24(4), pp. 10-22 (2004).

[Fa08] N. Farazmand, M. Fazeli, S.G. Miremadi, "FEDC: Control Flow Error Detection and Correction for embedded systems without program interruption". *ares, pp.33-38, 2008 Third International Conference on Availability, Reliability and Security, 2008.*

[Fr08] R. Fritzsche, G.Kemnitz, C. Siemers, "TEC: Time-Enhanced C – Erweiterung einer imperativen Programmiersprache um Zeitvorgaben". *Tagungsband Embedded Software Engineering, S. 427-430, Sindelfingen, Germany, Dezember 2008 (in German language).*

[Fr10] R. Fritzsche, C. Siemers, "Scheduling of Tme-Enhanced C (TEC)". *Proceedings of IEEE World Automation Conference 2010* (WAC 2010), *Kobe, Japan, September 2010, ISSN 2154-4824.*

[Ge01] M. Gergeleit, "A Monitoring-based Approach to Object-Oriented Real-Time Computing," *Otto-von-Guericke-Universität Magdeburg, Universitätsbibliothek, 2001, http://diglib.uni-magdeburg.de/Dissertationen/2001/margergeleit.pdf*

[He01] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. "Giotto: A time-triggered language for embedded programming", *Proceedings of the First International Workshop on Embedded Software* (EMSOFT), Lecture Notes in Computer Science 2211, Springer-Verlag, 2001, pp. 166-184.

[Ir06] K. Irrgang, J. Braunes, R.G. Spallek, S. Weisse, T. Gröger, "A new Concept for Efficient Use of Complex On-Chip Debug Solutions in SOC based Systems". *Embedded World 2006 Conference, pp. 215-223 (2006). Franzis Verlag, Poing, 2006, ISBN 3-7723-0143-6.*

[Le98] A. Leung, K. V. Palem, A. Pnueli, “TimeC: A time specification language for ILP processor compilation”, *Proceddings 12 of PART’98: The 5th Australasian Conference on Parallel And Real-Time Systems*, 1998, pp. 57-71.

[Li73] C.L. Liu, J.W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, *J. ACM*, vol. 20, no. 1, 1973.

[Ma88] A. Mahmood, E.J. McCluskey, “Concurrent Error DetectionUsing Watchdog Processors – A Survey”. *IEEE Transactions on Computers 37(2), pp. 160-174 (1988).*

[Mo97] Mok A. K., “A multiframe model for real-time tasks”, *IEEE Transactions on Software Engineering 23 (10)*, 1997, pp. 635-645.

[Na05] G. Naeser, L. Asplund, J. Furunäs, “Safety Chip – A Time Monitoring and Policing Device”. *SIGAda 05, pp. 63–68 (2005).*

[Ra04] A. Rajabzadeh, M. Mohandespour, G. Miremadi, "Error Detection Enhancement in COTS Superscalar Processors with Event Monitoring Features". *prdc, pp.49-54, 10th Pacific Rim International Symposium on Dependable Computing (PRDC'04), 2004*

[Ra05] A. Rajabzadeh, S.G. Miremadi, "A Hardware Approach to Concurrent Error Detection Capability Enhancement in COTS Processors," *prdc, pp.83-90, 11th Pacific Rim International Symposium on Dependable Computing (PRDC'05), 2005*

[Sa90] N.R. Saxena, E.J. McCluskey, “Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums”. *IEEE Transactions on Computers 39(4), pp. 554-559 (1990).*

[Si05] C. Siemers, R. Falsett, R. Seyer, K. Ecker, “Reliable Event-Triggered Systems for Mechatronic Applications,” ”. *The Journal of Systems and Software 77*, *Elsevier, 2005, pp. 17–26.*

[Si07] C. Siemers, “Coroutine Part 1+2”, *Embedded Software Engineering Report 2007*, Vogel Verlag, Würzburg, 2007, vol. 3. (in German language)